Titanium: A High Performance Dialect of Java

Kathy Yelick
http://www.cs.berkeley.edu/projects/titanium

U.C. Berkeley
Computer Science Division

Titanium Group

- Susan Graham
- Katherine Yelick
- Paul Hilfinger
- Phillip Colella (LBNL)
- Alex Aiken

- Greg Balls
- Peter McQuorquodale (LBNL)

- Andrew Begel
- Dan Bonachea
- David Gay
- Arvind Krishnamurthy
- Ben Liblit
- Carleton Miyamoto
- Chang Sun Lin
- Geoff Pike
- Luigi Semenzato (LBNL)
- Siu Man Yau

A Little History

- **Most parallel programs are written using explicit parallelism, either:**
  - Message passing with a SPMD model
    - Usually for scientific applications with C++/Fortran
    - Scales easily
  - Shared memory with a thread C or Java
    - Usually for non-scientific applications
    - Easier to program
- **Take the best features of both for Titanium**
  - Builds on ideas in Split-C, AC, and UPC
  - Safer language and more sophisticated implementation

Titanium

- **Take the best features of threads and MPI**
  - global address space like threads (programming)
  - SPMD parallelism like MPI (performance)
  - local/global distinction, i.e., layout matters (performance)
- **Based on Java, a cleaner C++**
  - classes, automatic memory management
  - compiled to C and then assembly (no JVM)
- **Optimizing compiler**
  - communication and memory optimizations
  - synchronization analysis
  - cache and other uniprocessor optimizations

Summary of Features Added to Java

- **Scalable parallelism:**
  - SPMD model of execution with global address space
- **Multidimensional arrays with iterators**
- **Checked Synchronization**
- **Immutable classes**
  - user-definable non-reference types for performance
- **Operator overloading**
- **Zone-based memory management**
- **Libraries**
  - Global communication
  - Distributed arrays
  - Fast bulk I/O

Lecture Outline

- **Language and compiler support for uniprocessor performance**
  - Immutable classes
  - Multidimensional Arrays
  - foreach
- **Language support for parallel computation**
- **Applications and application-level libraries**
- **Summary and future directions**

Java: A Cleaner C++

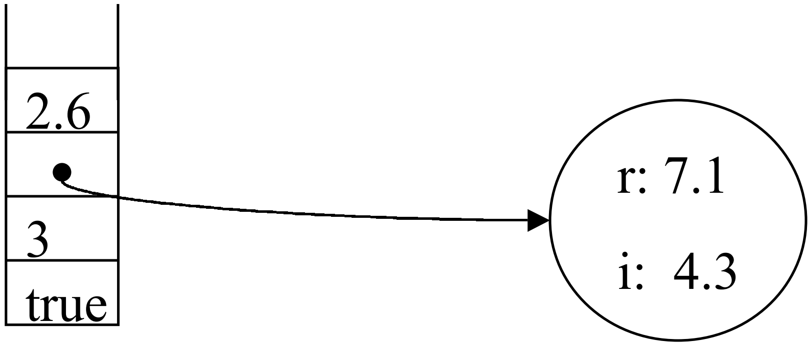
- **Java is an object-oriented language**
  - classes (no standalone functions) with methods
  - inheritance between classes
- **Documentation on web at java.sun.com**
- **Syntax similar to C++**

```
class Hello {
   public static void main (String [] argv) {
           System.out.println("Hello, world!");
   }
}
```

- **Safe: strongly typed, auto memory management**
- **Titanium is (almost) strict superset**

Java Objects

- **Primitive scalar types: boolean, double, int, etc.**
  - implementations will store these on the program stack
  - access is fast -- comparable to other languages
- **Objects: user-defined and standard library**
  - passed by pointer value (object sharing) into functions
  - has level of indirection (pointer to) implicit
  - simple model, but inefficient for small objects



Java Object Example

```
class Complex {
  private double real;
  private double imag;
  public Complex(double r, double i) {
       real = r; imag = i; }
  public Complex operator+(Complex c) {
       return new Complex(c.real + real,
                         c.imag + imag); }
  public double getReal {return real; }
  public double getImag {return imag; }
}
Complex c = new Complex(7.1, 4.3);
c = c + c;
```

Immutable Classes in Titanium

- **For small objects, would sometimes prefer**
  - to avoid level of indirection
  - pass by value (copying of entire object)
  - especially when immutable -- fields never modified
    - extends the idea of primitive values to user-defined values
- **Titanium introduces immutable classes**
  - all fields are final (implicitly)
  - cannot inherit from or be inherited by other classes
  - needs to have 0-argument constructor

Example of Immutable Classes

- **The immutable complex class nearly the same**

```
immutable class Complex {
    Complex () {real=0; imag=0; }
    ....
}
```

new keyword

Zero-argument constructor required

Rest unchanged. No assignment to fields outside of constructors.

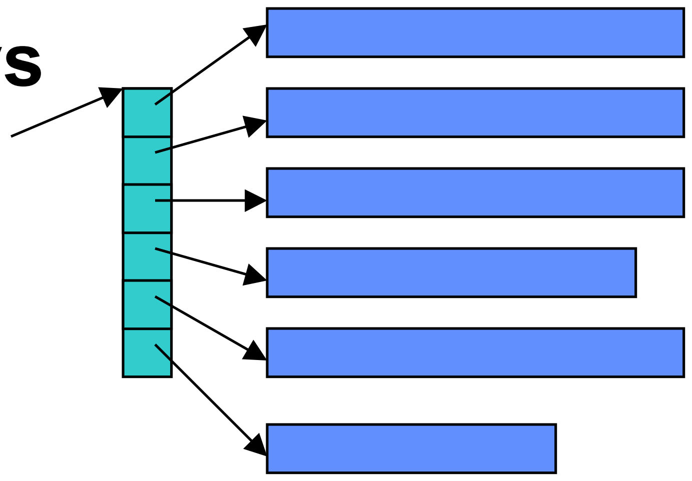
- **Use of immutable complex values**

```
Complex c1 = new Complex(7.1, 4.3);
Complex c2 = new Complex(2.5, 9.0);
c1 = c1 + c2;
```

Similar to structs in C in terms of performance

Arrays in Java

- **Arrays in Java are objects**
- **Only 1D arrays are directly supported**
- **Array bounds are checked**
  - Safe but potentially slow
- **Multidimensional arrays as arrays-of-arrays**
  - General, but slow



Multidimensional Arrays in Titanium

- **New kind of multidimensional array added**
  - Subarrays are supported (unlike Java arrays)
  - Indexed by Points (tuple of ints)
  - Constructed over a set of Points, called Domains
  - RectDomains (rectangular domains) are a special case
  - Points, Domains, RectDomains are immutable classes
- **Support for adaptive meshes and other mesh/grid operations**
  - e.g., can refer to the boundary region of an array

Point, RectDomain, Arrays in General

- **Points specified by a tuple of ints**

```
Point<2> lb = [1, 1];
Point<2> ub = [10, 20];
```

- **RectDomains given by 3 points:**
  - lower bound, upper bound (and stride)

```
RectDomain<2> r = [lb : ub];
```

- **Array declared by # dimensions and type**

```
double [2d] a;
```

- **Array created by passing RectDomain**

```
a = new double [r];
```

Simple Array Example

- **Matrix sum in Titanium**

```
Point<2> lb = [1,1];
Point<2> ub = [10,20];
RectDomain<2> r = [lb,ub];

double [2d] a = new double [r];
double [2d] b = new double [1:10,1:20];
double [2d] c = new double [lb:ub:[1,1]];

for (int i = 1; i <= 10; i++)
   for (int j = 1; j <= 20; j++)
     c[i,j] = a[i,j] + b[i,j];
```

No array allocation here

Syntactic sugar

Optional stride

Naïve MatMul with Titanium Arrays

```
public static void matMul(double [2d] a,
    double [2d] b, double [2d] c) {
  int n = c.domain().max()[1]; // square
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
           c[i,j] += a[i,k] * b[k,j];
         }
      }
   }
}
```

Array Performance Issues

- **Array representation is fast, but access methods can be slow, e.g., bounds checking, strides**
- **Compiler optimizes these**
  - common subexpression elimination
  - eliminate (or hoist) bounds checking
  - strength reduce: e.g., naïve code has 1 divide per dimension for each array access

- **Currently +/- 20% of C/Fortran for large loops**
- **Future: small loop and cache optimizations**

Unordered iteration

- **All of these optimizations require loop analysis**
- **Compilers can do this for simple operations, e.g., matrix multiply, but hard in general**
- **Titanium adds unordered iteration on rectangular domains -- gives user more control**

```
foreach (p within r) { ... }
```

  - p is a Point new point within the foreach body
  - r is a previously-declared RectDomain

Laplacian Example

- **Simple example of using arrays and foreach**

```
Domain<2> interior = A.domain().shrink(1);
Point<2> dx = [1,0];
Point<2> dy = [0,1];
foreach (p in interior) {
  L[p] = 4*a[p] - a[p+dx] - a[p-dx]
                - a[p+dy] - a[p-dy];
}
```

Better MatMul with Titanium Arrays

```
public static void matMul(double [2d] a,
       double [2d] b, double [2d] c) {
  foreach (ij within c.domain()) {
    double [1d] aRowi = a.slice(1, ij[1]);
    double [1d] bColj = b.slice(2, ij[2]);
    foreach (k within aRowi.domain()) {
      c[ij] += aRowi[k] * bColj[k];
    }
  }
}
```

Current performance: comparable to 3 nested loops in C

Sequential Performance

Ultrasparc:	C/C++/ FORTRAN	Java Arrays	Titanium Arrays	Overhead
DAXPY	1.4s	6.8s	1.5s	7%
3D multigrid	12s		22s	83%
2D multigrid	5.4s		6.2s	15%
MatMul	1.8s		2.2s	22%

Pentium II:	C/C++/ FORTRAN	Java Arrays	Titanium Arrays	Overhead
DAXPY	1.8s		2.3s	27%
3D multigrid	23.0s		20.0s	-13%
2D multigrid	7.3s		5.5s	-25%

Compares to naïve C code; neither compiler does cache blocking (yet).

Lecture Outline

- **Language and compiler support for uniprocessor performance**
- **Language support for parallel computation**
  - SPMD execution
  - Barriers and single
  - Explicit Communication
  - Implicit Communication (global and local references)
  - More on Single
  - Synchronized methods and blocks (as in Java)
- **Applications and application-level libraries**
- **Summary and future directions**

SPMD Execution Model

- **Java programs can be run as Titanium, but the result will be that all processors do all the work**
- **E.g., parallel hello world**

```
class HelloWorld {
  public static void main (String [] argv) {
    System.out.println(‘’Hello from proc ‘’ +
                       Ti.thisProc());
    }
 }
```

- **Any non-trivial program will have communication and synchronization**

SPMD Execution Model

- A common style is compute/communicate
- E.g., in each timestep within particle simulation with gravitation attraction

```
read all particles and compute forces on mine
Ti.barrier();
write to my particles using new forces
Ti.barrier();
```

SPMD Model

- **All processor start together and execute same code, but not in lock-step**
- **Basic control done using**
  - Ti.numProcs() total number of processors
  - Ti.thisProc() number of executing processor
- **Sometimes they take different branches**

```
if (Ti.thisProc() == 0) { ….. do setup ..… }
System.out.println(‘’Hello from ‘’ + Ti.thisProc());
double [1d] a = new double [Ti.numProcs()];
```

Barriers and Single

- Common source of bugs is barriers or other global operations inside branches or loops

```
barrier, broadcast, reduction, exchange
```

- A "single" method is one called by all procs

```
public single static void allStep(….)
```

- A "single" variable has same value on all procs

```
int single timestep = 0;
```

- Single annotation on methods (also called "sglobal") is optional, but useful to understanding compiler messages.

Explicit Communication: Broadcast

- **Broadcast is a one-to-all communication**

```
broadcast <value> from <processor>
```

- **For example:**

```
int count = 0;
int allCount = 0;
if (Ti.thisProc() == 0) count = computeCount();
allCount = broadcast count from 0;
```

- **The processor number in the broadcast must be single; all constants are single.**
- **The allCount variable could be declared single.**

Example of Data Input

- Same example, but reading from keyboard
- Shows use of Java exceptions

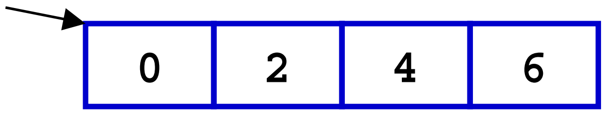
```
int single count = 0;
int allCount = 0;
if (Ti.thisProc() == 0)
  try {
     DataInputStream kb = new DataInputStream(System.in);
     myCount = Integer.valueOf(kb.readLine()).intValue();
  } catch (Exception e) {
    System.err.println(``Illegal Input'');
allCount = myCount from 0;
```

Explicit Communication: Exchange

- **To create shared data structures**
  - each processor builds its own piece
  - pieces are exchanged (for object, just exchange pointers)
- **Exchange primitive in Titanium**

```
int [1d] single allData;
allData = new int [0:Ti.numProcs()-1];
allData.exchange(Ti.thisProc()*2);
```

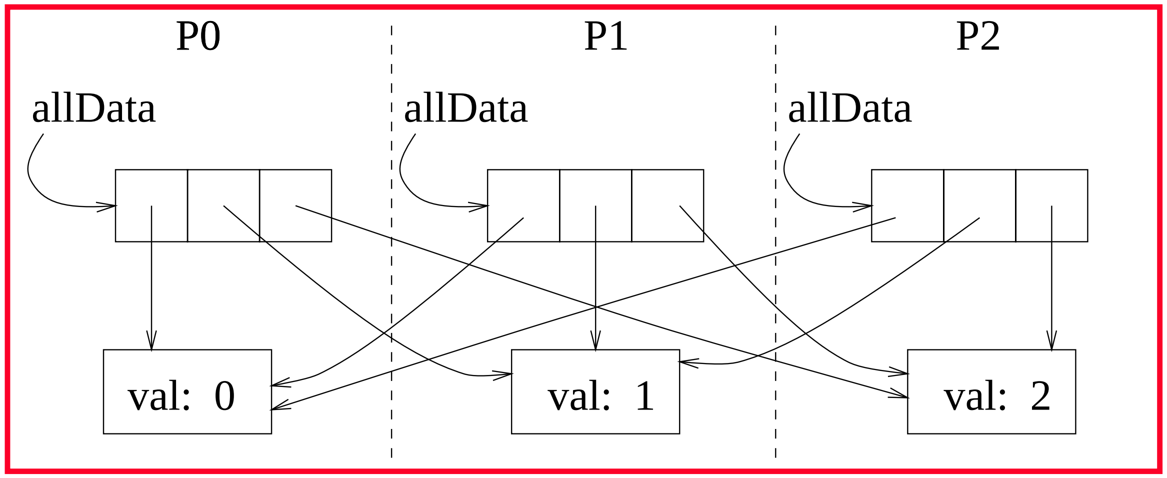
- **E.g., on 4 procs, each will have copy of allData:**



Exchange on Objects

- **More interesting example:**

```
•   class Boxed {
•       public Boxed (int j) {
•         val = j;
•       }
•       public in val;
•   }
```



```
• Object [1d] single allData;
• allData = new Object [0:Ti.numProcs()-1];
• allData.exchange(new Boxed(Ti.thisProc());
```

Distributed Data Structures

- Build distributed data structures with arrays:

```
RectDomain <1> single allProcs = [0:Ti.numProcs-1];
RectDomain <1> myParticleDomain = [0:myPartCount-1];
Particle [1d] single [1d] allParticle =
                   new Particle [allProcs][1d];
Particle [1d] myParticle =
                   new Particle [myParticleDomain];
allParticle.exchange(myParticle);
```

- Now each processor has array of pointers, one to each processor's chunk of particles

More on Single

- **Global synchronization needs to be controlled**
  - if (this processor owns some data) {
  -     compute on it
  -     barrier
  - }
- **Hence the use of “single” variables in Titanium**
- **If a conditional or loop block contains a barrier, all processors must execute it**
  - conditions in such loops, if statements, etc. must contain only single variables

Single Variable Example

- **Barriers and single in N-body Simulation**

```
class ParticleSim {
   public static void main (String [] argv) {
   int single allTimestep = 0;
   int single allEndTime = 100;
   for (; allTimestep < allEndTime; allTimestep++){
      read all particles and compute forces on mine
      Ti.barrier();
      write to my particles using new forces
      Ti.barrier();
    }
  }
}
```

- **Single methods inferred; see David Gay's work**

Use of Global / Local

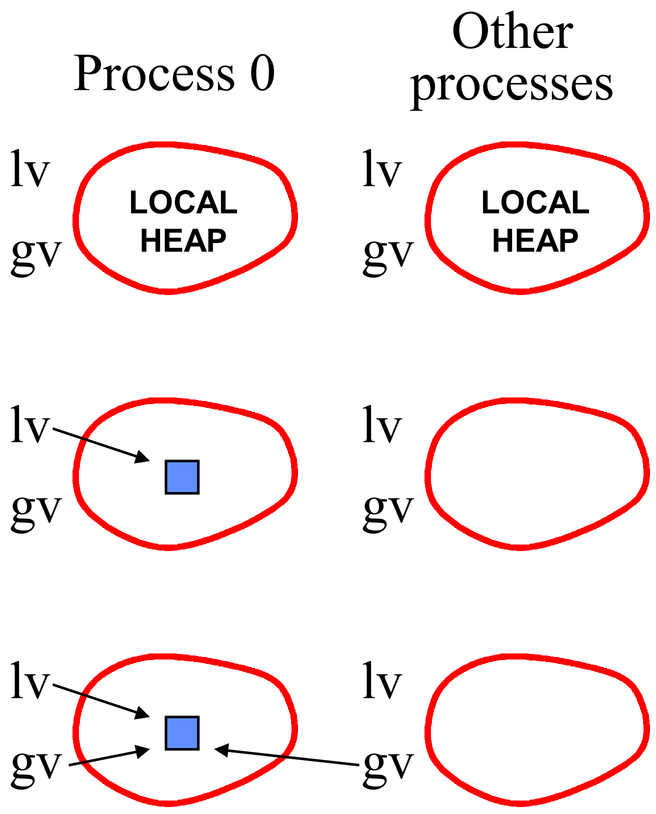
- **As seen, references (pointers) may be remote**
  - easy to port shared-memory programs
- **Global pointers are more expensive than local**
  - True even when data is on the same processor
  - Use local declarations in critical sections
- **Costs of global:**
  - space (processor number + memory address)
  - dereference time (check to see if local)
- **May declare references as local**

Global Address Space

- **Processes allocate locally**
- **References can be passed to other processes**

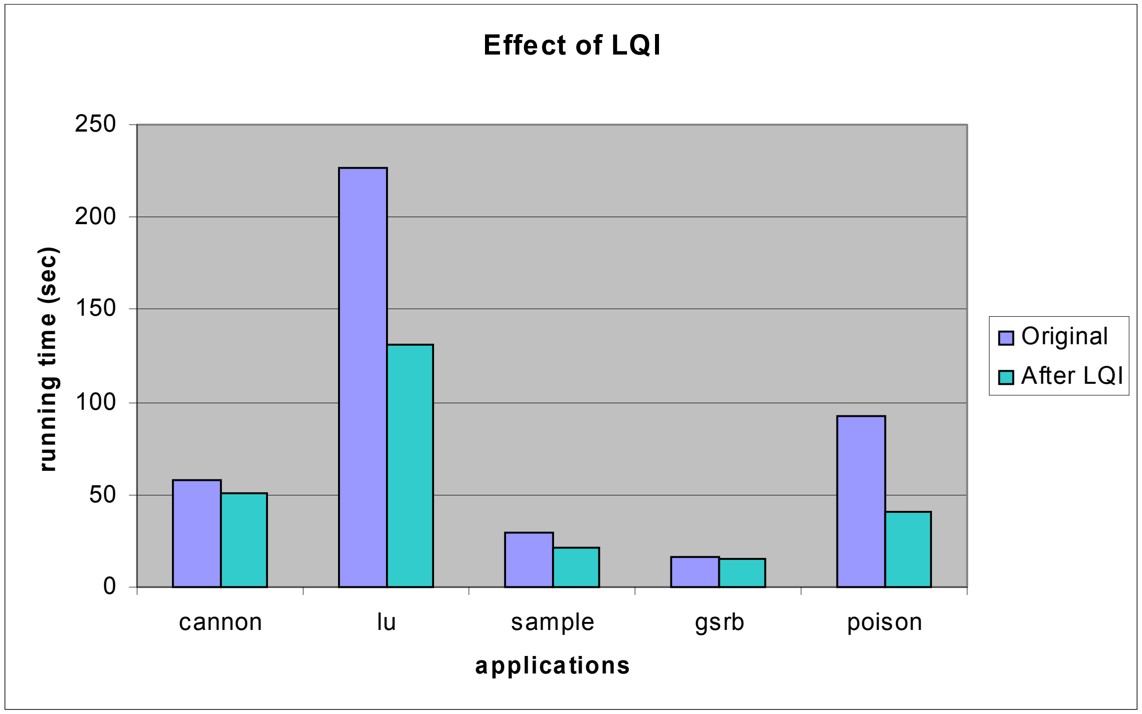
```
Class C { int val;.. }
C gv;          // global pointer
C local lv; // local pointer

if (thisProc() == 0) {
      lv = new C();
}
gv = broadcast lv from 0;
gv.val = ..; // full
.. = gv.val; // functionality
```



Local Pointer Analysis

- **Compiler can infer many uses of local**
  - See Liblit's work on Local Qualification Inference



- **Data structures must be well partitioned**

Region-Based Memory Management

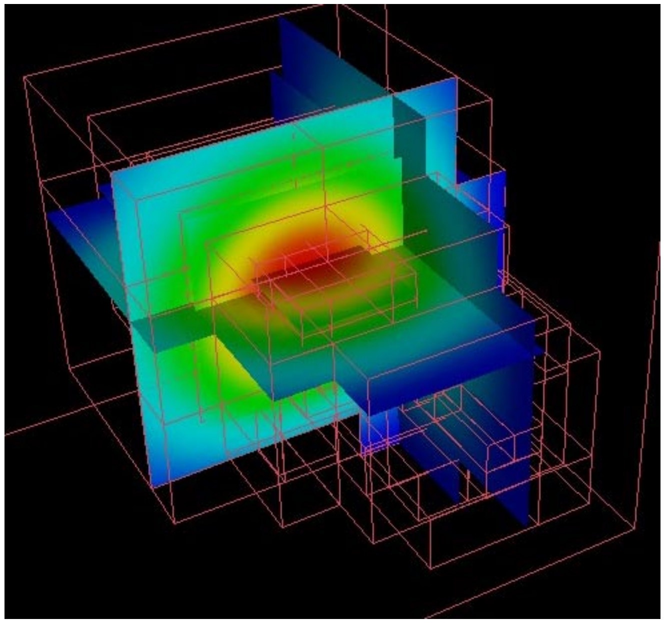
```
PrivateRegion r = new PrivateRegion();
For (int j = 0; j < 10; j++) {
   int[] x = new ( r ) int[j + 1];
   work(j, x);
}
try { r.delete; }
catch (RegionInUse oops) {
    system.out.println("failed to delete");
  }
}
```

Lecture Outline

- **Language and compiler support for uniprocessor performance**
- **Language support for parallel computation**
- **Applications and application-level libraries**
  - AMR overview
  - AMR and uniform grid algorithms in Titanium
  - Several smaller benchmarks
    - MatMul, LU, FFT, Join, Sort, EM3d
  - Library interfaces
    - PETSc, Metis,
- **Summary and future directions**

Block-Structured AMR

- **Algorithms for many rectangular, grid-based computations are**
  - communication intensive
  - memory intensive

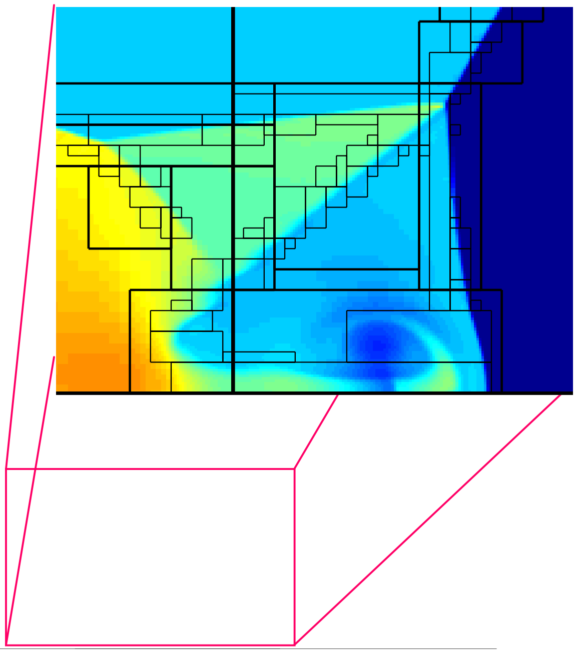


- **AMR makes these harder**
  - more small messages
  - more complex data structures
  - most of the programming effort is debugging the boundary cases
  - locality and load balance trade-off is hard

Algorithms for AMR

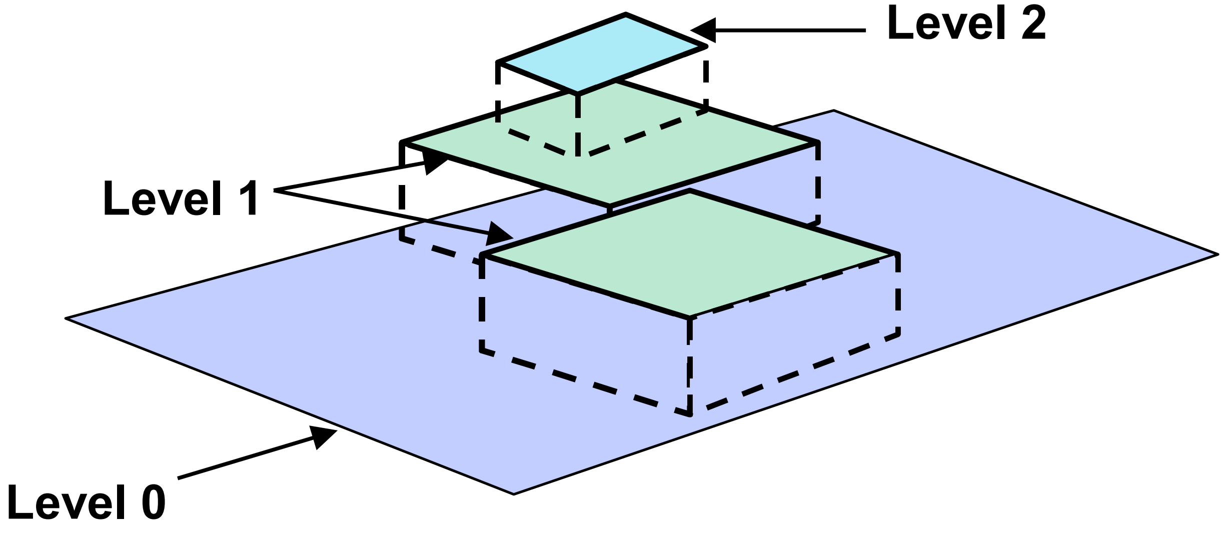
- **Existing algorithms in Titanium**
  - 3D AMR Poisson solver
  - 3D AMR Gas dynamics
  - Domain-decomposition MLC Poisson
- **Under development**
  - Self-gravitating gas dynamics (3D AMR)
    - For stellar collapse, etc.
  - Immersed boundary method (3D, non-adaptive)
    - Peskin and MacQueen's method for heart model, etc.
  - Embedded boundaries
    - Simulation of bio-MEMs devices and cellular level modeling
  - Project Idea:
    - Multiblock Java code with self-scheduling.   Contact me, yelick@cs.
    - Evaluation of and proposal for general domains.
- **All joint with Colella's group at LBNL**

3D AMR Gas Dynamics



3D AMR Poisson

- **Poisson Solver [Semenzato, Pike, Colella]**
  - finite domain
  - variable coefficients
  - multigrid across levels



- **Currently synthetic grids, no grid generation**
- **Under construction**
  - reengineered to interface with hyperbolic solver
  - including mesh generation

MLC for Finite-Differences

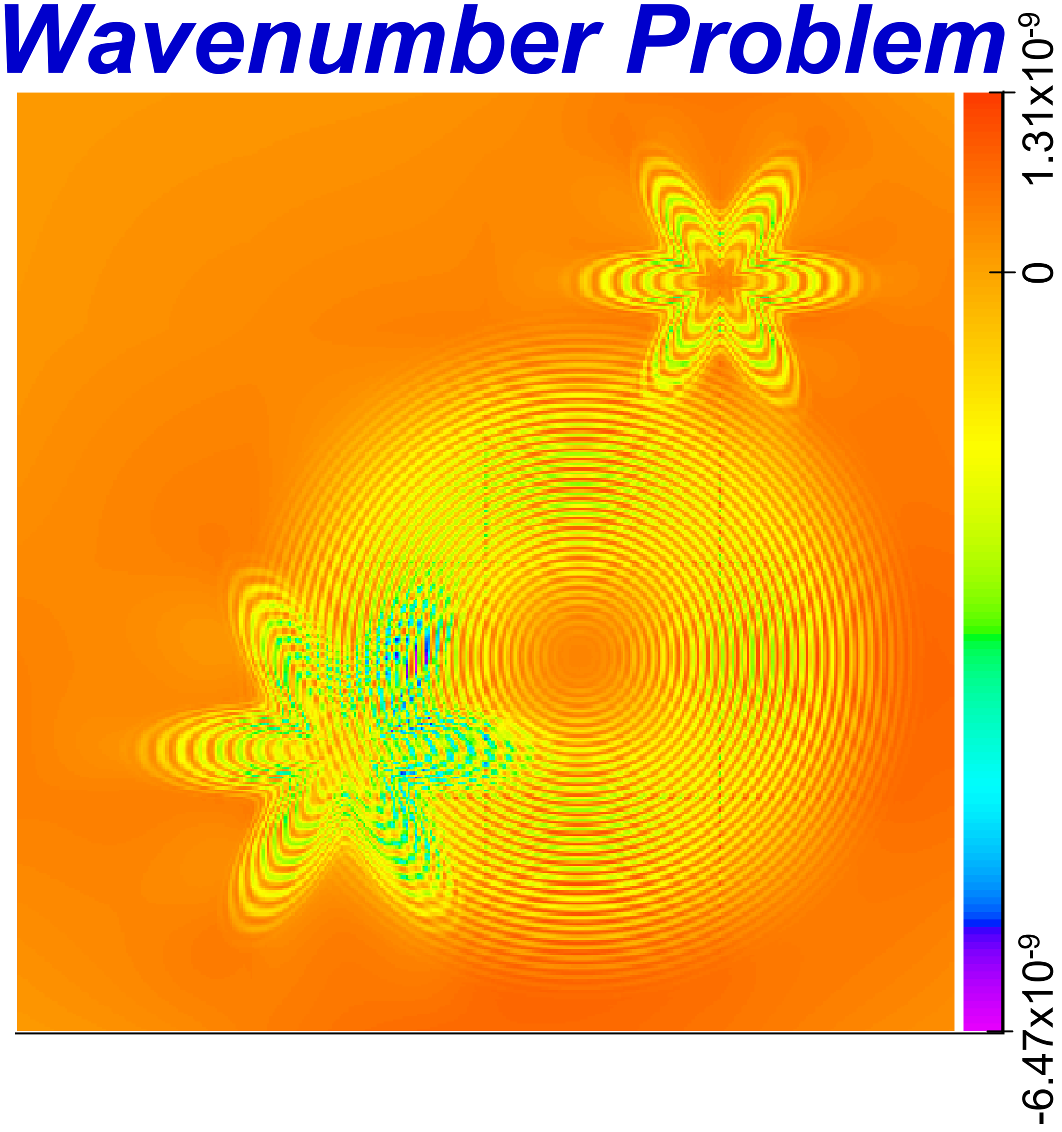
- **Poisson solver with infinite domains [Colella, Balls]**
  - Uses a Method of Local Corrections (MLC)
  - Currently non-adaptive and 2D
  - Supports only constant coefficients
- **Uses 2-level, domain decomposition approach**
  - Fine-grid solutions are computed in parallel
  - Information transferred to a coarse-grid and solved serially
  - Fine-grid solutions is computed using boundary conditions from the coarse grid
- **Future work includes 3D Adaptive version**

MLC for Finite-Differences

- **Features of the method**
  - Solution is still second-order accurate
  - Accuracy depends only weakly on the coarse-grid spacing
- **Scalability**
  - No communication during fine-grid solves
  - Single communication step (global all-to-all)
  - coarse grid work is serial (replicated), but relatively small
- **Future work: extend to 3D and adaptive meshes**
  - Project idea: extension to 3D: see Greg Balls, gballs@cs

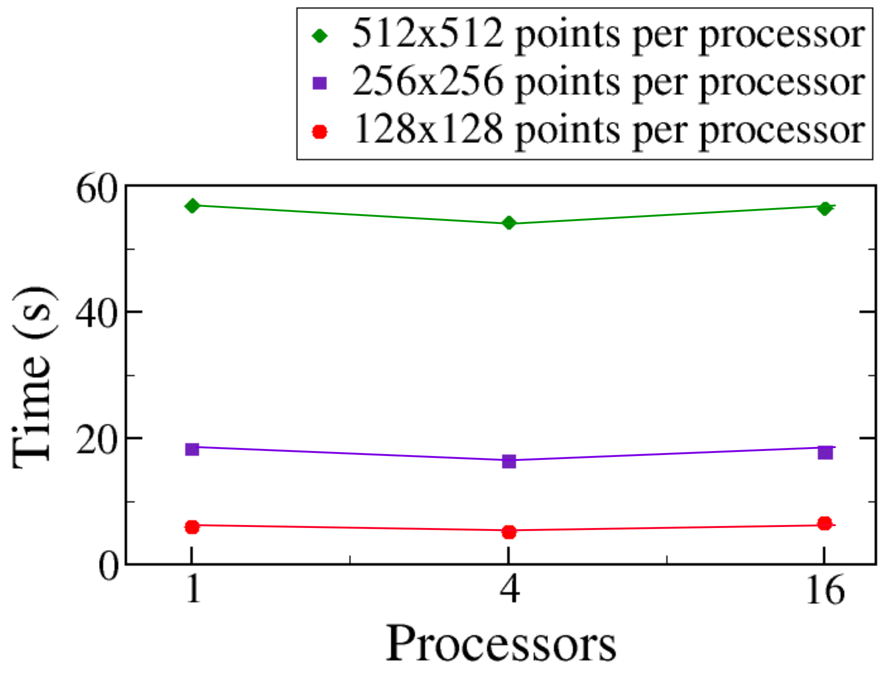
Error on High-Wavenumber Problem

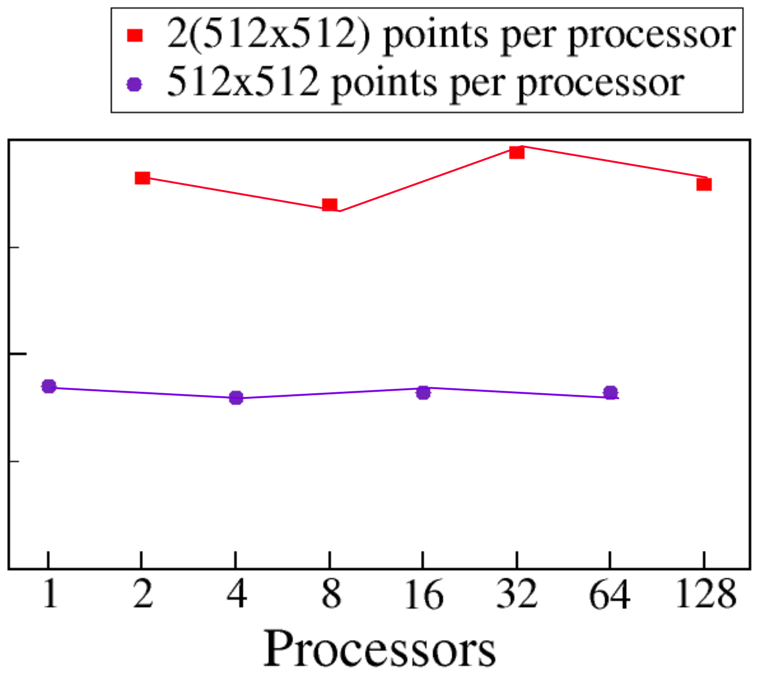
- **Charge is**
  - 1 charge of concentric waves
  - 2 star-shaped charges.
- **Largest error is where the charge is changing rapidly. Note:**
  - discretization error
  - faint decomposition error
- **Run on 16 procs**



Scalable Poisson Solver (MLC)

- **Communication performance is low (< 5%)**
- **Scaled speedup experiments are nearly ideal (flat)**

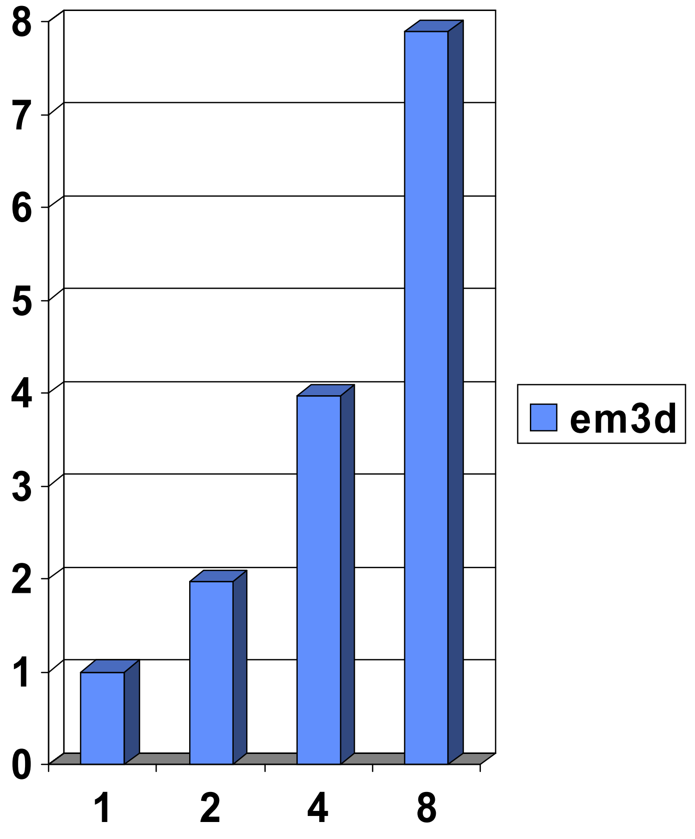




Titanium

Unstructured Mesh Kernel

- EM3D: Relaxation on a 3D unstructured mesh
- Speedup on Ultrasparc SMP
- Simple kernel: mesh not partitioned.



Calling Other Languages

- **We have built interfaces to**
  - PETSc : scientific library for finite element applications
  - Metis: graph partitioning library
- **Two issues with cross-language calls**
  - accessing Titanium data structures (arrays) from C
    - possible because Titanium arrays have same format on inside
  - having a common message layer
    - Titanium is built on lightweight communication

Lecture Outline

- **Language and compiler support for uniprocessor performance**
- **Language support for parallel computation**
- **Applications and application-level libraries**
- **Summary and future directions**
  - Implementation

Implementation

- **Strategy**
  - Titanium into C
  - Solaris or Posix threads for SMPs
  - Lightweight communication for MPPs/Clusters
- **Status: Titanium runs on**
  - Solaris or Linux SMPs and uniprocessors
  - Berkeley NOW
  - SDSC Tera, SP2, T3E (NERSC and NPACI)
  - SP3 (and IBM SP Power3) port underway

Titanium Summary

- **Performance**
  - close to C/FORTRAN + MPI on limited class of problems
- **Portability**
  - develop on uniprocessor, then SMP, then MPP/Cluster
- **Safety**
  - as safe as Java, extended to parallel framework
- **Expressiveness**
  - easier than MPI, harder than threads
- **Compatibility, interoperability, etc.**
  - no gratuitous departures from Java standard

Using Titanium

- **On machines in the CS Division**

  ```
  /srs/titanium/*/bin/tcbuild file.ti
  ```

  - Solaris 2.6 and Linux supported; need to mount this filesystem
- **On NERSC t3e use:**

  ```
  /u/mp215/miyamoto/tc-1.44/tcbuild/tcbuild file.ti
  ```

- **On SP2 contact:** `cjlin@cs.berkeley.edu`
- **For documentation, source code, see the home page**
  - http://www.cs.berkeley.edu/projects/titanium
- **Documentation includes**
  - Language reference, terse but complete
  - Tutorial, incomplete
- **For problems or questions:**

  `titanium-group@cs.berkeley.edu`

Future Plans

- **Improved compiler optimizations for scalar code**
  - large loops are currently +/- 20% of Fortran
  - working on small loop performance
- **Packaged solvers written in Titanium**
  - Elliptic and hyperbolic solvers, both regular and adaptive
- **New application collaboration**
  - Peskin and McQueen (NYU) with Colella (LBNL)
  - Immersed boundary method, currently use for heart simulation, platelet coagulation, and others

Backup Slides

Other Language Extensions

Java extensions for expressiveness & performance

- Operator overloading
- Zone-based memory management
- Foreign function interface

The following is not yet implemented in the compiler

- Parameterized types (aka templates)

Consistency Model

- **Titanium adopts the Java memory consistency model**
- **Roughly: Access to shared variables that are not synchronized have undefined behavior.**
- **Use synchronization to control access to shared variables.**
  - barriers
  - synchronized methods and blocks